July 27, 1999

Flow analysis models for interprocedural program slicing algorithms

Arun Lakhotia, Jean-Christophe Deprez, and Shreyash S. Kame

The Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504
(318) 482-6766, -5791 (Fax)
arun@cacs.usl.edu

Abstract

We present flow analysis models for dependence graph based algorithms for (a) computing program slices and (b) computing summary information needed for slicing. Our models benefits typical of declarative specification. First, they show that algorithms for solving flow analysis problems may be applied to computing slices and summary information. Second, they make it explicit that SDG-based algorithms presented in the literature are essentially instances of the well-known worklist algorithm for solving flow analysis problems. Third, the models, being declarative, concisely express what is being computed, instead of focussing on how it is being computed. The models are more concise than the algorithms and provide a formal foundation for comparing different algorithms for the same problem. Third, with the existence of tools to generate analyzers from flow models, our models remove the necessity to present algorithms. As a demonstration, we present flow analysis model for a new interprocedural slicing algorithm that combines the computation of summary information and the computation of slice. That the new algorithm is equivalent to previous algorithms can be demonstrated by showing that their flow models are isomorphic.

Index Terms: Program slicing, flow analysis models, worklist algorithm, system dependence graph.

1 Introduction

A slice of a program with respect to a program point p consists of all statements of the program that might affect the behavior of the program observed at p; the program point p is said to be the *slicing criterion*[*]. The problem of computing slices of a program has received considerable attention over the last two decades because of its applicability in system generation, debugging, verifying requirements [32], program integration [13], restructuring [17, 22, 23], testing [10], program comprehension [6, 11], and reuse [5]. The interest and significance of program slicing is further outlined by the existence of three survey papers [4, 16, 30] on the subject and a recent special issue [12].

Program slicing algorithms may also be classified as follows [4, 16, 30]:

1. **Dataflow equation based**—influenced by Weiser's model of a program slice as a solution of data flow equations.
2. **Graph reachability based**—influenced by Ottenstein & Ottenstein's model of the problem as a graph-reachability problem [25].
3. **Information flow relation based**—Bergeretti and Carré's model of a slice in terms of information flow relations derived from a program using a syntax-directed approach [2].
4. **Denotational semantics based**—influenced by the use of denotational semantics to express program analyzers [31].

[*] This definition is slightly different from the original definition of program slice introduced by Weiser [33].

In all, but the graph-reachability based approaches, what constitutes a program slice may be modelled *declaratively* as a system of equations (more generally, inequations). The solution of these equations constitutes a slice. Algorithms for solving these equations, being known as part of a larger body of work, are therefore not always discussed, except to enumerate the method of finding the solution. In contrast, the graph-reachability based slicing algorithms are presented *operationally*, as traversal of program or system dependence graph (SDG) [14, 28]. The problem being modelled as a graph-reachability problem, researchers provide (a) algorithms for constructing the necessary graphs and (b) algorithms for traversing the graph to identify the statements belonging to a slice.

In this paper, we present *declarative* models [24] for several graph-reachability based slicing algorithms. In our models the underlying graph reachability problem is specified as a set of constraints. These constraints satisfy the conditions of monotone flow analysis framework and hence are termed flow models. Unlike monotone flow models for problems like reaching definitions or constant propagations which are developed on a control flow graph (CFG), our flow analysis models use SDG.

The construction of a SDG requires solving the reaching definition problem using CFG, a problem whose flow analysis models may be found in the literature [24]. It also requires computing certain summary information whose computation has been presented in the literature algorithmically [14, 28]. We present flow models for the computation of summary information as well.

Our flow analysis models of program slicing and summary information offer the following benefits, typical of declarative specification:

1. The models make it explicit that algorithms for solving flow analysis problems are directly applicable to the problem of computing program slices. More generally, the models show that graph-reachability problems may be mapped to flow analysis problems. This is the converse of Reps, Horwitz, and Sagiv's finding that certain classes of flow analysis problems may be mapped to the graph-reachability problem [27].
2. The models make it explicit that the known SDG-based slicing algorithms are special cases of the worklist algorithms for solving flow analysis problems [14, 28]. Besides the worklist algorithm, there are several other algorithms for solving such problems, such as algorithms using strongly connected components, or specialized algorithms when the underlying graph is reducible, and Gaussian elimination algorithms [24]. The advantages or disadvantages of using these other flow analysis algorithms for program slicing has not been investigated and may be worthy of further research.
3. The models enable concise description of different slicing algorithms and provide a formal foundation to prove their equivalence. For instance, we present flow analysis model of a new interprocedural slicing algorithm that computes summary edges at the same time that a slice is computed. Since the computation of the summary edges dominates the cost of interprocedural slicing, computing this information only if it is needed may offer computational speed up especially if slicing is performed in an environment where the program may be changing. That the new algorithm is equivalent to previous algorithms can be demonstrated by proving that the models for the two algorithms are isomorphic.
4. There already exists at least one tool, the Program Analyzer Generator (PAG), that given a flow model of a problem can automatically generate a program that solves that problem [24]. Thus, having developed the models, the task of developing the algorithm becomes routine.

The rest of the paper is organized as follows. Section 2 summarizes the program and system dependence graphs, and also gives a quick overview of flow analysis models and the worklist algorithm to solve them. Section 3 gives flow analysis models of several intraprocedural and interprocedural slicing problem. It also gives models for computation of summary edges, a type of edge in the system dependence graph, and a model for a new interprocedural slicing algorithm. Section 4 concludes the paper. Some commonly used definitions needed for this paper are presented in Appendix A. Proof of equivalence of models of two interprocedural slicing algorithm is presented in Appendix B.

2 Preliminaries

This section presents some background knowledge needed for the rest of the paper. The next subsection gives an overview of program and system dependence graph. It is followed by a subsection on flow analysis models.

Dependence graphs A system dependence graph (SDG) [14] encodes the data, control, and call dependence relations between statements* of a program in a simple procedural language consisting of assignment, if-then-else, while-do, procedure call, entry, and return statements. The parameters to a procedure call are simple variables passed by value-reference. There is a special procedure *main* from which execution is initiated.

A SDG consists of a collection of procedure dependence graphs (PDG) (a variation of program dependence graph [19, 25]). There is one PDG per procedure in the program encoding the control and data dependence relations within the procedure. These graphs contain vertices representing *call-sites* and procedure *entry* points. The SDG has *call edges* connecting the call-sites in a PDG to the entry point in the PDG of the procedure called at that site.

For each call-site, a PDG also contains two vertices for every actual parameter in the procedure call. An *actual-in* vertex to represent the transfer of value of the actual parameter to an intermediate variable used to send the input to the procedure. An *actual-out* vertex to represent transfer of the final value of the parameter from an intermediate variable to the actual parameter.

Analogously, every formal parameter of a procedure is represented in a PDG by two vertices. A *formal-in* vertex representing the transfer of value to the formal parameter from the intermediate variable assigned to in the *actual-in* vertex. A *formal-out* vertex representing the transfer of value from the formal parameter to the intermediate variable used at the *actual-out* vertex.

The pairs of intermediate variables used to communicate the initial and final values of a parameter to/from the procedure entry are unique. A SDG contains two types of edges, *parameter-in* and *parameter-out*, to represent the data dependence between the *actual-in* to *formal-in* and *formal-out* to *actual-out* vertices.

Since there is one PDG per procedure in a SDG, the actual-in (actual-out) vertices for the same parameter from different calls to the same procedure are depended upon by (depend on) the same formal-in (formal-out) vertex. The SDG also contains edges between the actual parameter vertices of the same procedure call and between the formal parameter vertices at the procedure entry. These edges, termed *summary edges*, summarize the dependence between the arguments of the procedure call as a result of executing the procedure.

Domain model for SDG We introduce a collection of domains to represent the nodes and edges of a SDG. The domain `N` denotes the set of all nodes of a SDG. The elements of this domain are partitioned into the following subsets:

`FormalIn`: the set of all *formal-in* nodes and *entry* nodes.
`FormalOut`: the set of all *formal-out* nodes.
`ActualIn`: the set of all *actual-in* nodes and *call-site* nodes.
`ActualOut`: the set of all *actual-out* nodes.
`Other`: the set of nodes not in any of the above subsets†.

We use the same domain to represent *formal-in* nodes and *entry* nodes because these nodes require identical treatment for the problem being studied. The same holds for *actual-in* nodes and *call-site* nodes. The correspondence between the `ActualIn` nodes and `FormalIn` nodes, similarly `ActualOut` nodes `FormalOut` nodes, is important in specifying the models. The following functions provide the necessary mappings.

C : `ActualIn` $\rightarrow$ `FormalIn`

C : `ActualOut` $\rightarrow$ `FormalOut`

R : `ActualIn` $\rightarrow$ `FormalOut` $\rightarrow$ `ActualOut`

The unction C is polymorphic. It maps an `ActualIn` node to the corresponding `FormalIn` node of the procedure called, and an `ActualOut` node to the corresponding `FormalOut` node. An *entry* node is considered to be the `FormalIn` node corresponding to an `ActualIn` representing a *call-site*.

* It also has another dependence relation called the def-order dependence which is not relevant for slicing. This relation is therefore ignored in this paper.

† For the presentation of this paper we do need to separately classify other nodes. They are implicitly classified due to the classification of edges.

The function R maps a `FormalOut` node to its corresponding `ActualOut` node at the call site containing the `ActualIn`.

The above functions are based on the assumption that each call-site calls a unique procedure. This assumption is violated when a language has procedure-valued variables. In such case, the co-domain of function C may be modified to be a set of either `FormalIn` or `FormalOut` nodes. Corresponding changes would be needed in the usage of this function too.

The domain `E` denotes the set of all edges of a SDG. Its elements are further partitioned into the following subsets:

`Intra`: *Control* and *flow* edges between vertices of the same PDG.
`Summary`: Edges from `ActualIn` nodes to `ActualOut` nodes representing the dependence created due to a procedure call.
`Call`: *Call* and *parameter-in* edges from `ActualIn` nodes to `FormalIn` nodes.
`Return`: *Parameter-out* edges from `FormalOut` nodes to `ActualOut` nodes.

We use the following notation to simply the presentation.

Notation: Let $f : A \rightarrow B$. Then $\overline{f} : 2^A \rightarrow 2^B$ is defined as $\overline{f}(X) = \{f(x) \mid x \in X\}$.

Flow analysis models We now present an overview of flow analysis models for program analysis. The description has been simplified to capture the concepts necessary to present models of program slicing algorithms. More general description of flow analysis models may be found elsewhere [24]. Definitions of terms used below are presented in Appendix A.

A flow analysis model contains the following components:

1. D : A *semantic domain*, a complete lattice satisfying the ascending chain condition.
2. $G = (N, E)$: A directed graph, where N is a set of nodes representing statements (or computational units) and $E \subseteq N \times N$ is a set of edges representing some semantic relation between two computational units[‡].
3. A set of equations *specifying* the analysis. The equations are usually recursive and may be abstracted as: $Analysis = f(Analysis)$, where $Analysis : N \rightarrow D$, i.e., $Analysis$ associates to every computational unit a value from the set D, the *semantic domain*. The equations are further constrained in that the analysis of a computational unit is described only in terms of its successor in the directed graph G, $\forall n.Analysis(n) = f_n(Y), Y = [Analysis(y) \mid (n, y) \in E]$.

Notation: The expression $[x \mid P(x)]$ denotes a sequence of elements satisfying $P(x)$. Every use of a specific expression $[x \mid P(x)]$ denotes the same sequence.

The least fixed point of the set of equations is usually the desired solution, which may be computed by the worklist algorithm in Figure 1. That the algorithm computes the desired analysis is guaranteed if function f is monotone.

The system of equations in which the analysis at a node is defined as a function of its successors is called a *backward* flow model. Such equations are suitable for describing the *backward* slicing problem, the context of the discussion. A *forward* flow model is constructed when the analysis at a node is defined as a function of its predecessors. The woklist algorithm may trivially be adapted to solve forward flow models by replacing all instances of the expression $(x, y) \in E$ by $(y, x) \in E$. Thus, the ideas presented in the paper are equally applicable to the *forward* slicing problem.

As is done in the next section, an analysis may sometimes be specified as a set of inequations, instead of equations, such as:

$$Analysis(x) \sqsupseteq e_1$$
$$\vdots$$

[‡] Flow analysis models are most commonly used with the control flow graph representation of a program, which imposes additional path constraints. These constraints are not necessary for using the worklist algorithm [24].

INPUT: A directed graph (N, E).

A system of equations, $Analysis(n) = f_n(Y), Y = [Analysis(y) \mid (n, y) \in E], \forall n \in N$

OUTPUT: Solution for the equations

METHOD:

```
- - Initialization of Worklist and Solution
for all n ∈ N do
    Solution[n] := f_n([⊥, ⊥, ...]);
    if Solution[n] ⊐ ⊥ then
        Worklist := Worklist ∪ { n }
- - Iteration (update Worklist and Solution)
while Worklist ≠ φ do
    Remove a node n from Worklist;
    foreach x such that (x, n) ∈ E do
        new := Solution[x] ⊔ f_x(Y), Y = [Solution(y) | (x, y) ∈ E]
        if Solution[x] ⊏ new then
            Solution[x] := new;
            Worklist := Worklist ∪ { x };
```

Figure 1 Worklist based algorithm for solving flow equations.

$$Analysis(x) \sqsupseteq e_k$$

where all the constraints have the same left hand side. The least solution of this system of inequations is also the least solution of $Analysis(x) = e_1 \sqcup \cdots \sqcup e_k$, [24, Chapter 6]. Though, Nielson *et al.* give algorithms to solve a system of inequations, we assume that the inequations may be mapped to the equivalent (with respect to their least solution) equations and solved using the worklist algorithm of Figure 1. We do so because the slicing algorithms in the literature are instances of this worklist algorithm.

All the system of constraints we present satisfy the conditions required to ensure that their least fixed point can be computed by the worklist algorithm. Hence we treat the set of recursive constraints to define the corresponding functions.

3 Flow analysis models for program slicing

We now present flow analysis models for program slicing. We start with model for intraprocedural slicing, then give models for interprocedural slicing for Horwitz, Reps, and Binkley's two pass algorithm [14] and Lakhotia's one pass algorithm [20]. That a worklist algorithm for a flow model is equivalent to the graph reachability based algorithm it models may be determined by instantiating the worklist algorithm. We present such an instantiation for flow models of intraprocedural slicing. Algorithm for interprocedural models may be instantiated similarly.

That such flow models may be developed for problems other than slicing is demonstrated by giving the flow models for computing summary edges. Though these edges are helpful in computing a slice, as part of system dependence graph they are also helpful for other algorithms based on SDG [3, 21]. The worklist algorithm for our models for summary edge computation is computationally equivalent, in solution and computational complexity, to the fast algorithm of Reps *et al.* [28].

```
INPUT:    A directed graph (N, E).
          A system of equations, Slice_A(n), as described in text
          With slicing criteria S
OUTPUT:   Solution for the equations
METHOD:   - - Initialization of Worklist and Solution
          for all n ∈ N do
              Solution[n] := if n ∈ S then ⊤ else ⊥;
              if Solution[n] = ⊤
                  Worklist := Worklist ∪ { n };
          - - Iteration (update Worklist and Solution)
          while Worklist ≠ φ do
              Remove a node n from Worklist;
              foreach x such that (x, n) ∈ E do
                  new := ⊤
                  if Solution[x] ⊏ new then
                      Solution[x] := new;
                      Worklist := Worklist ∪ { x };
```

Figure 2 Worklist based intraprocedural slicing algorithm. This algorithm is created by specializing the algorithm of Figure 1 by the constraints for intraprocedural slicing.

Intraprocedural slicing The following inequations give a flow model for intraprocedural slicing.
Domain: $D_A = \{\top, \bot\}$, where $\bot \sqsubseteq \top$
Constraints: $Slice_A : N \rightarrow D_A$

$$Slice_A(x) \sqsupseteq \bigsqcup_{x \rightarrow y \in \text{Intra}} Slice_A(y)$$
$$Slice_A(x) \sqsupseteq \top, \ x \in S$$

where S is the slicing criterion. In the above constraints that a node is in the slice is represented by associating to the value $\top$ to that node. According to the constraints a node x is in the slice (1) if any of its successor is in the slice or (2) if it is in the slicing criterion S.

The worklist algorithm of Figure 1 when instantiated for the above semantic domain and constraints yields the algorithm in Figure 2, which is the same algorithm as that proposed by Ottenstein & Ottenstein[25]. A node x is in the slice if on termination of the algorithm `Solution[`x`]` is $\top$, the least fixed point solution of the constraints.

Computing Interprocedural slices An interprocedural slicing algorithm based on reachability over the system dependence graph was first proposed by Horwitz, Reps, and Binkley [14]. This algorithm computes the slice in two passes. In the first pass the `Return` edges are excluded from traversal and in the second pass the `Call` edges are excluded. This two pass traversal ensures that information from a call site is not inadvertently propagated to another call site of the same procedure.

Figure 3 (a) gives the flow model for the slicing algorithm of Horwitz, Reps, and Binkley's algorithm [14]. Horwitz *et al.*'s two-pass algorithm solves the constraints for $Slice_{H1}$ in the first pass and the constraints for

$Slice_{H2}$ in the second pass. The constraints may be read as follows. The first pass, $Slice_{H1}$, identifies a node to be in the slice if either it is in the slicing criterion or if any of its `Intra`, $Summary$, or `Call` successor is in the slice. The second pass, $Slice_{H2}$, identifies a node to be in the slice if either it was identified to be in the slice in first pass or if any of its `Intra`, $Summary$, or `Return` successor is in the slice. Thus, node x is in the slice if $Slice_{H2}(x)$ is equal to $\top$. Each pass of Horwitz *et al.*'s slicing algorithm is essentially an instance of the Worklist algorithm of Figure 1.

The two sets of constraints can also be solved in one pass by solving them simultaneously. Lakhotia's one pass algorithm does essentially that [20]. But it solves the constraints of Figure 3 (b). According to this model a node x is in the slice iff $Slice_L(x)$ is either equal to $\top$ or β. The constraints state that a node is in the slice with value (not less than) $\top$ if it is in the slicing criterion. Otherwise, a node is in the slice if its `Intra` or $Summary$ successor is, and the node takes at least the same value as its successors. A node is also in the slice if any of its `Call` successors is in slice with the value $\top$. The function μ ensures that if a node is not placed in slice if the value at its `Call` successor is β. A node may also be in the slice if any of its `Return` successor is in slice. In such case, the function φ ensures that the node gets atleast the value β irrespective of the value at the `Call` successor. In the least fixed point solution a node is assigned the smallest value that is greater than or equal to the values needed to satisfy all the constraints.

That, Lakhotia's algorithm is equivalent to Horwitz et al.'s algorithm can be determined by showing that the two models are isomorphic, as formulated below:

Definition: $D_H = D_{H1} \times D_{H2}$ and $Slice_H : N \rightarrow D_H$, such that $Slice_H(x) = (Slice_{H1}(x), Slice_{H2}(x))$.

Lemma: $Slice_H$ and $Slice_L$ are isomorphic.

Proof: In Appendix B.

Computing summary information We now present flow constraints for computing summary edges of a SDG. The constraints are written in a form that enumerates their correspondence with Reps *et al.'s* fast algorithm for computing summary edges [28].

Constraints: $PathEdge : N \rightarrow 2^{\texttt{FormalOut}}$

$$PathEdge(x) \supseteq \{x\}, x \in \text{ FormalOut}$$

$$PathEdge(x) \supseteq \bigcup_{x \rightarrow y \in \text{Intra}} PathEdge(y)$$

$$PathEdge(x) \supseteq \bigcup_{x \rightarrow y \in \text{Call}} \overline{PathEdge}\left(\overline{R(x)}(PathEdge(y))\right)$$

Constraints: $SummaryEdge : \texttt{ActualIn} \rightarrow 2^{\texttt{ActualOut}}$

$$Summary(x) = \overline{R(x)}(PathEdge(C(x)))$$

The constraints for $PathEdge$ state the following. The $PathEdge$ function associates a set of `FormalOut` nodes to every node in the graph. That a `FormalOut` node y is in $PathEdge(x)$ implies that in the SDG (without summary edges) there is a realizable-path from y to x. A realizable path is a path in which the call and return edges are paired to represent a valid execution path. The constraints may be read as follows. There is a realizable path from every `FormalOut` node to itself. If there is an `Intra` edge $x \rightarrow y$ and there is a realizable path from y to a `FormalOut` node z then there is a realizable path from node x to node z. If there is a `Call` edge $x \rightarrow y$ (so y is implicitly a `FormalIn` node) to a `FormalOut` node z then there is realizable path from x to every node in $PathEdge(R(x)(z))$, where the expression $R(x)(z)$ gives the actual-out node corresponding to the formal out node z at the call-site containing x.

The $Summary$ function associates a set of `ActualOut` nodes to every `ActualIn` nodes. That an `ActualOut` node y is in $Summary(x)$ implies that there is a reachable path from the `ActualIn` node x to the `ActualOut` node y. Though not expressed as such, the constraint essentially says that y is in $Summary(x)$ if there is a reachable path from $C(x)$ to $C(y)$.

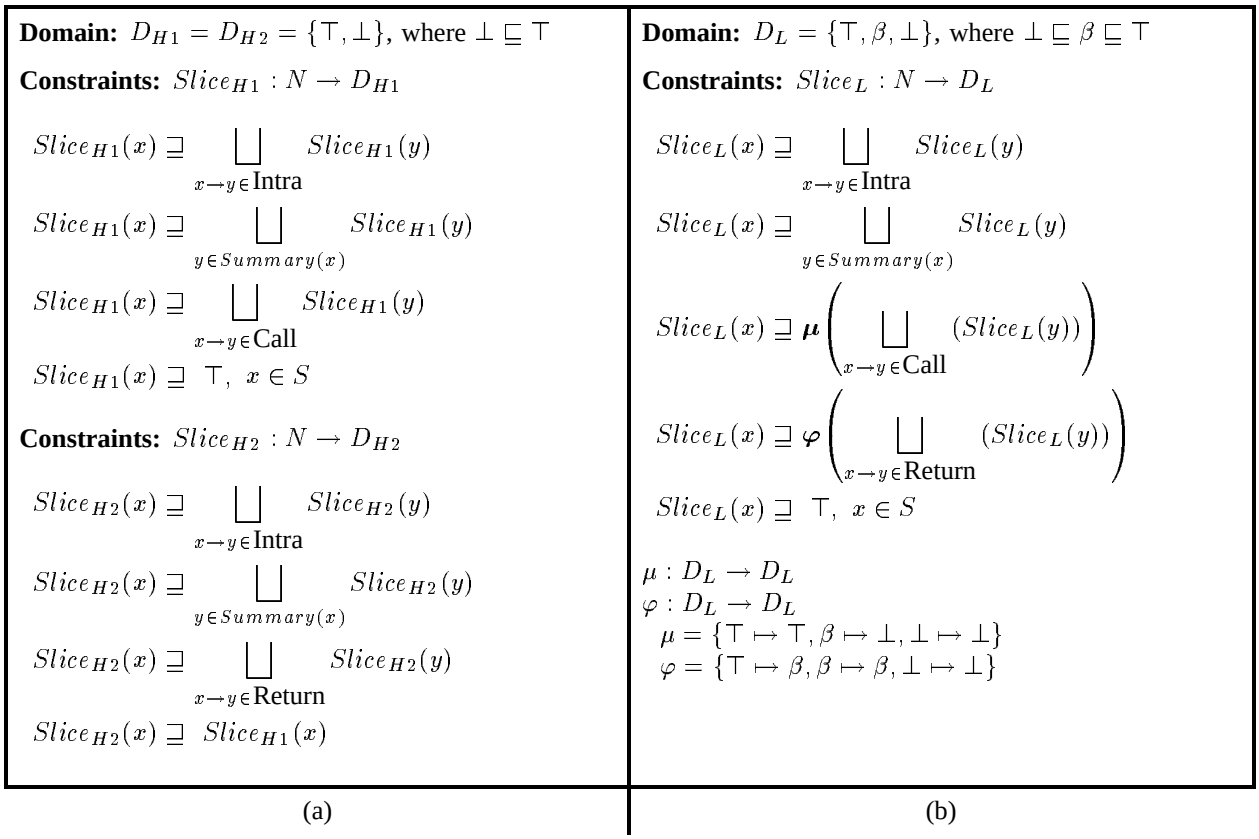

Figure 3 Flow constraints for interprocedural slicing. The symbol $S \subseteq 2^N$ gives the set of nodes in the slicing criterion. (a) Flow model for Horwitz, Reps, and Binkley's two pass algorithm [14]. (b) Flow model for Lakhotia's one pass algorithm [20].

The above $PathEdge$ and $Summary$ functions are represented in Reps *et al.'s* algorithm by binary relations with the same name. We refer to Reps *et al.*'s relations as $PathEdge_R$ and $Summary_R$. On termination of the respective algorithms on the same input these structures will be related as follows:

Lemma: $PathEdge_R = \{(x, d) \mid x \in N, d \in PathEdge(x)\}$ and $Summary_R = \{(x, d) | x \in N, d \in Summary(x)\}$.

Proof: Left to the reader.

Computing summary information during slicing Horwitz *et al.*'s [14] and Reps *et al.*'s [28] algorithms compute summary information exhaustively. Since summary information is interprocedural its computation can be expensive, whether computed using Reps *et al.*'s fast algorithm, which is an instance of the worklist algorithm for previous flow constraints. When slicing is used for debugging, as also for other applications in a program development environment, it may not be computationally prudent to compute the summary edges exhaustively. It may be preferred to compute the summary information only if it is needed.

The following flow models for interprocedural slicing combines the computation of summary information with the computation of slice. The summary information is computed only for those actual-out nodes that are in the slice.

Domain: $D_I = \top + 2^{\texttt{FormalOut}} + \bot$[§],

such that $\forall \theta \subseteq \texttt{FormalOut}.\bot \sqsubseteq \theta \sqsubseteq \top$, and $\forall \theta_1, \theta_2 \subseteq \texttt{FormalOut}.\theta_1 \sqsubseteq \theta_2$ iff $\theta_1 \subseteq \theta_2$.

§ D_I is a *disjoint union* of $\top$, $2^{\texttt{FormalOut}}$, and $\bot$. For the sake of brevity the constraints are written as though D_I is a *union* of these domains.

Constraints: $Slice_I : N \rightarrow D_I$

$$Slice_I(x) \sqsupseteq \bigsqcup_{x \rightarrow y \in \text{Intra}} Slice_I(y)$$
$$Slice_I(x) \sqsupseteq \bigsqcup_{x \rightarrow y \in \text{Call}} \Pi(x)(Slice_I(y))$$
$$Slice_I(x) \sqsupseteq \bigsqcup_{x \rightarrow y \in \text{Return}} \Phi(x)(Slice_I(y))$$
$$Slice_I(x) \sqsupseteq \top,\ x \in S$$

where S is the slicing criterion and

Π : `ActualIn`$\rightarrow D_I \rightarrow D_I$

$\Pi = \lambda x.\{\top \mapsto, \theta \mapsto \overline{Slice_I}(\overline{R(x)}(\theta)), \bot \mapsto \bot\}$

Φ : `FormalOut`$\rightarrow D_I \rightarrow D_I$

$\Phi = \lambda x.\{\top \mapsto \{x\}, \theta \mapsto \{x\}, \bot \mapsto \bot\}$

The constraints for $Slice_I$ combine the constraints for $Slice_L$ and $PathEdge$. This is reflected in the elements of domain D_I. This domain may trivially be mapped to D_L, the domain of $Slice_L$, by mapping all non-empty set of `FormalOut` nodes θ to β. Besides, θ also represents elements of the domain of $PathEdge$. According to the above constraints a node x is in the slice if $Slice(x)$ is either $\top$ or a non-empty set of `FormalOut` nodes θ.

The functions Π and Φ are key to understanding the above constraints. The function Π transforms information flowing from a `FormalIn` node y to an `ActualIn` node x. A $\top$ at a `FormalIn` node is propagated as a $\top$ to the `ActualIn` node. However, a set θ, representing a set of `FormalOut` nodes, reaching y is not propagated to x. Instead, the information at `ActualOut` nodes corresponding to the `FormalOut` nodes θ is propagated to x. The function Φ transforms information flowing from an `ActualOut` node y to a `FormalOut` node x. Whatever information, except $\bot$, reaches an `ActualOut` node y the set $\{x\}$ is propagated to the `FormalOut` node x.

The following proposition relates the constraints for $Slice_L$ and $Slice_I$.

Proposition: $\forall x.Slice_L(x) = \top \Leftrightarrow Slice_I(x) = \top$ and $Slice_L(x) = \beta \Leftrightarrow \exists \theta \subseteq$`FormalOut`$.\theta \neq \phi \wedge Slice_I(x) = \theta$.

$PathEdge$ and $Slice_I$ are related as follows:

Proposition: $\forall x.Slice_I(x) = \theta \Rightarrow \theta \subseteq PathEdge(x)$.

The above propositions may be proved following structure similar to the proof in Appendix B. The details of the proof are left for the reader.

4 Related works

A detailed comparison with literature on interprocedural analysis models, problems, and algorithms is beyond the scope of paper.

There has been a considerable amount of work on program slicing. Exhaustive survey of the literature may be found in the reviews by Binkley and Gallagher [4], and Kamkar [16], and Tip [30]. Based on the taxonomy used in these surveys the constraints we have provided are for *static, backward* slice. Our constraints can trivially be adapted for *static, forward* slice as well. The computation of *dynamic* forward and backward slicing requires different information, which too is usually represented as graphs. But the dynamic slicing algorithms are not pure graph reachability algorithms [1, 18]. It may be worth investigating if these algorithms can be modelled using flow constraints.

Forgács and Gyimóthy have proposed an algorithm for computing summary edges by taking advantage of strongly connected components in a call-graph [9]. Their improvement is analogous to improving the worklist based flow analysis algorithms using strongly connected components [24]. The difference, however, is that a call

graph is an abstraction of a system dependence graph. Thus, the SCC-worklist does not directly map to Forgács and Gyimóthy's improvement.

Reps, Horwitz, and Sagiv have shown that a large class of interprocedural dataflow analysis problems can be transformed to graph-reachability problems [27]. They take a dataflow analysis problem whose solution is of the form $Analysis : N \rightarrow 2^V$, equivalently $Analysis : N \rightarrow V \rightarrow Boolean$, where N is the set of nodes of the underlying graph and V is a finite set (such as set of variables), and transform it to a problem of the form $Analysis' : N \times V \rightarrow Boolean$. After transformation the new graph has $N \times V$ nodes. A node is assigned a true value only if it can be reached from the start node. Reps *et al.* provide a compact representation of the flow function at each node which enables solving the encoded problem by computing all the nodes that can be reached from a start node.

Reps has suggested a method for transforming the exhaustive version of summary edges computation into an equivalent demand problem by applying the magic-set transformation [26]. Horwitz, Reps, and Sagiv's have proposed dynamic programming based demand algorithm for computing summary information [15]. Our flow constraints for computing summary edges, $Summary$, satisfy the requirements of Duesterwald, Gupta, and Soffa's framework for demand-driven interprocedural analysis [7]. Thus, a demand algorithm for the problem follows directly from their framework. Alternatively, the constraints may be solved using Fecht & Siedl's fast algorithm to compute solutions on demand [8, 29].

The flow constraints wherein the computation of summary edges is combined with the computation of slice, $Slice_I$, are new. The worklist algorithm resulting from these constraints is a contribution of this paper.

Binkley's SDG based interprocedural constant propagation algorithm [3] and Lakhotia's SDG based algorithm for constructing call multigraph have also been presented operationally. The computations performed by these algorithm may be abstracted as flow models similar to those presented here.

5 Conclusions

In the literature dependence graph based program slicing algorithms have been presented *operationally*. This paper shows that these algorithms are essentially instances of the classic worklist algorithm for data flow analysis. This relation was previously not obvious because of the absence of *declarative* models of the underlying problem. The flow analysis models for program slicing we present enumerate this relationship. That these flow relations are presented on a dependence graph, as against a control flow graph, implies that flow models for other dependence graph based algorithms can be developed similarly.

The flow models we present are a preferred expression of the underlying computation because they are declarative, they only express *what* is being computed. The constraints represented by the models may be solved using generic dataflow analysis algorithms. The slicing algorithms in the literature have so far are instances of only the worklist based dataflow analysis algorithm. Our models make it explicit that other dataflow analysis algorithms, such as elimination-based algorithms, T1–T2 analysis, reducible graph based algorithm, may also be used for program slicing. Investigation of the advantages and disadvantages of using these algorithms for program slicing may be worthy of further experimental research.

Bibliography

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the ACM Fourth Symposium on Testing, Analysis, and Verification*, Oct. 1991.

[2] J.-F. Bergeretti and B. Carré. Information flow and data-flow analysis of **while**–programs. *ACM Trans. Prog. Lang. Syst.*, 7(1):37–61, 1985.

[3] D. Binkley. Interprocedural constant propagation using dependence graphs and a data-flow model. *Lecture Notes in Computer Science*, 786:374–388, 1994. Proceedings of the Fifth International Conference on Compiler Construction.

[4] D. Binkley and K. Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[5] A. Cimitile, A. De Lucia, and M. Munro. A specification driven slicing process for identifying reusable functions. *Software Maintenance: Research and Practice*, 8:145–178, 1996.

[6] A. De Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviors through program slicing. In *Fourth IEEE Workshop on Program Comprehension*, pages 9–18, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[7] E. Duesterwald, R. Gupta, and M. L. Soffa. A practial framework for demand-driven interprocedural data flow analysis. *ACM Trans. Prog. Lang. Syst.*, 19(6), Nov. 1997.

[8] C. Fecht and H. Seidl. An even faster solver for general systems of equations. In *Proceedings of Static Analysis, Third Symposium (SAS'96)*, pages 189–204, 1996. Lecture Notes in Computer Science, Vol. 1145, Springer 1996.

[9] I. Forgács and T. Gyimóthy. An efficient interprocedural slicing method for large programs. In *Proceedings of SEKE'97, the 9th International Conference on Software Engineering & Knowledge Engineering, Madrid, Spain*, pages 279–287, 1997.

[10] M. Harman and S. Danicic. Using program slicing to simplify testing. *Journal of Software Testing, Verification, and Reliability*, 5:143–162, 1995.

[11] M. Harman and S. Danicic. Amorphous program slicing. In *Fifth IEEE International Workshop on Program Comprehension (IWPC'97)*, pages 70–79, Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[12] M. Harman and K. Gallagher, editors. *Special issue on Program Slicing, Journal of Information and Software Technology*, volume 40. Elsevier, 1998.

[13] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *ACM Trans. Prog. Lang. Syst.*, 11(3):345–387, July 1989.

[14] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Prog. Lang. Syst.*, 12(1):26–60, 1990.

[15] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *SIGSOFT '95: Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 104–115, 1995. ACM SIGSOFT Software Engineering Notes 20, 4 (1995).

[16] M. Kamkar. An overview and comparative classification of program slicing techniques. *Journal of Systems and Software*, 31:197–214, 1995.

[17] H. S. Kim, I. S. Chung, and Y. R. Kwon. Restructuring programs through program slicing. *International Journal of Software Engineering and Knowledge Engineering*, 4(3):349–368, Sept. 1994.

[18] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29:155–163, 1988.

[19] D. J. Kuck, Y. Muraoka, and S. Chen. On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up. *IEEE Transactions on Computers*, C-12(12), Dec. 1972.

[20] A. Lakhotia. Improved interprocedural slicing algorithm. Technical Report CACS-TR-92-5-8, University of Southwestern Louisiana, Lafayette, Nov. 1992.

[21] A. Lakhotia. Constructing call multigraphs using dependence graphs. In *ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages (POPL'93)*, pages 273–284, Jan. 1993.

[22] A. Lakhotia. DIME: A direct manipulation environment for evolutionary development of software. In *Proceedings of the International Workshop on Program Comprehension (IWPC'98)*, pages 72–79, Los Alamitos, CA, June 1998. IEEE Computer Society Press.

[23] A. Lakhotia and J.-C. Deprez. Restructuring programs by tucking statements into functions. *Journal of Information and Software technology*, 40(11-12):677–689, Nov. 1998.

[24] F. Nielson, H. R. Nielson, and C. Hankin. Principles of program analysis. Preprint, 1998.

[25] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. *ACM SIGPLAN Notices*, 19(5), May 1984.

[26] T. Reps. Solving demand versions of interprocedural analysis problems. In P. Fritzson, editor, *Proceedings of the Fifth International Conference on Compiler Construction, (Edinburgh, Scotland, April 7-9, 1994)*, pages 389–403, New York, NY, 1994. Springer-Verlag. Lecture Notes in Computer Science, Vol. 786.

[27] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 49–61, Jan. 1995.

[28] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. *ACM SIGSOFT Software Engineering Notes*, 19:11–20, Dec. 1994. SIGSOFT'94: Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering.

[29] H. Seidl and C. Fecht. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. *Nordic Journal of Computing*, 5:304–329, 1998. (Also appeared in the Proceedings of 7th European Symposium on Programming, ESOP'1998, Lecture Notes in Computer Science, Vol. 1381, Springer, 1998, pp. 90-104).

[30] F. Tip. A survey of program slicing techniques. *J. Program. Lang.*, 3:121–181, 1995.

[31] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 107–119, 1991.

[32] M. Weiser. *Program Slicing: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, The University of Michigan, Ann Arbor, Michigan, 1979.

[33] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, 1984.

Appendix A

Definition: $(D, \sqsubseteq)$ is a *partially ordered set* iff $\sqsubseteq$ is a reflexive, transitive, and anti-symmetric binary relation over D.

For convenience we state that D is a partially ordered set whenever the relation $\sqsubseteq$ is obvious from context. For a pair of elements d_1, $d_2 \in D$, the relation $d_1 \sqsubseteq d_2$ is read as d_1 is *less than or equal* to d_2, or conversely d_2 is *greater than or equal* to d_1.

Definition: An element $d \in D$ is an *upper bound* of a subset Y of D iff d is greater than or equal to every element of Y. A element d is a *least upper bound* of Y iff it is an upper bound of Y and is less than or equal to all other upper bounds of Y. The least upper bound of a subset Y of D is denoted by $\bigsqcup Y$, or $d_1 \sqcup d_1$ when $Y = \{d_1, d_2\}$.

Definition: An element $d \in D$ is a *lower bound* of a subset Y of D iff d is less than or equal to every element of Y. It is a *greatest lower bound* of Y iff it is a lower bound of Y and is greater than or equal to all other lower bounds of Y. The greatest lower bound of a subset Y of D is denoted by $\bigsqcap Y$ or $d_1 \sqcap d_1$ when $Y = \{d_1, d_2\}$.

A subset Y of a partially ordered set D may not always have a least upper bound or a greatest lower bound, but when they do exist they are unique.

Definition: A partially ordered set D is a *complete lattice* iff all its subsets have least upper bounds and greatest lower bounds. A complete lattice has a *least* element, denoted $\bot$, and a *greatest* element, denoted $\top$, where $\bot = \bigsqcup \phi = \bigsqcap D$ and $\top = \bigsqcap \phi = \bigsqcup D$.

Definition: A subset Y of a partially ordered set D is a *chain* iff its elements can be totally ordered using $\sqsubseteq$.

Definition: A partially ordered set D satisfies the *ascending chain condition* iff every chain in D as a least upper bound in D.

Definition: A function $f : A \rightarrow B$ is *monotonic* iff for all $x, y \in A$, $x \sqsubseteq_A y \Rightarrow f(x) \sqsubseteq_B f(y)$.

Definition: Let D be a complete lattice. An element $x \in D$ is a *fixed point* of a function $f : D \rightarrow D$ iff $f(x) = x$. Let $Fix(f)$ denote all the fixed points of the function f. Then $\bigsqcap Fix(f)$, denoted $lfp(f)$, gives the *least fixed point* of f.

Tarski's Fixed Point Theorem: If D is a complete lattice satisfying the ascending chain condition and $f : D \rightarrow D$ is a monotone function then the least fixed point of f exists and is defined as $\bigsqcup\{f^i(\bot) \mid i \geq 0\}$, where $f^i = f \circ f \circ \cdots \circ f$, i times.

Appendix B

This appendix proves that the interprocedural slicing models in Figures 3 (a) and (b) are isomorphic.

Definition: $D_H = D_{H1} \times D_{H2}$ and $Slice_H : N \rightarrow D_H$, such that $Slice_H(x) = (Slice_{H1}(x), Slice_{H2}(x))$.

Lemma: $Slice_H$ and $Slice_L$ are isomorphic.

Proof: The proof follows by showing that the constraints for $Slice_H$ can be transformed into the constraints for $Slice_L$ using fold/unfold equivalence preserving transformations.

Step 1. In the constraints for $Slice_{H2}$ unfold the constraints for $Slice_{H1}$.

$$Slice_{H2}(x) \sqsupseteq \bigsqcup_{x \to y \in \text{Intra}} Slice_{H1}(y) \sqcup \bigsqcup_{x \to y \in \text{Intra}} Slice_{H2}(y)$$

$$Slice_{H2}(x) \sqsupseteq \bigsqcup_{y \in Summary(x)} Slice_{H1}(y) \sqcup \bigsqcup_{y \in Summary(x)} Slice_{H2}(y)$$

$$Slice_{H2}(x) \sqsupseteq \bigsqcup_{x \to y \in \text{Call}} Slice_{H1}(y)$$

$$Slice_{H2}(x) \sqsupseteq \bigsqcup_{x \to y \in \text{Return}} Slice_{H2}(y)$$

$$Slice_{H2}(x) \sqsupseteq \top, x \in S$$

Step 2. Unfold $Slice_{H1}$ and $Slice_{H2}$ in the definition of $Slice_H$.

$$Slice_H(x) \sqsupseteq \left(\bigsqcup_{x \to y \in \text{Intra}} Slice_{H1}(y), \bigsqcup_{x \to y \in \text{Intra}} Slice_{H1}(y) \sqcup \bigsqcup_{x \to y \in \text{Intra}} Slice_{H2}(y) \right)$$

$$Slice_H(x) \sqsupseteq \left(\bigsqcup_{y \in Summary(x)} Slice_{H1}(y), \bigsqcup_{y \in Summary(x)} Slice_{H1}(y) \sqcup \bigsqcup_{y \in Summary(x)} Slice_{H2}(y) \right)$$

$$Slice_H(x) \sqsupseteq \left(\bigsqcup_{x \to y \in \text{Call}} Slice_{H1}(y), \bigsqcup_{x \to y \in \text{Call}} Slice_{H1}(y) \right)$$

$$Slice_H(x) \sqsupseteq \left(\bot, \bigsqcup_{x \to y \in \text{Return}} Slice_{H2}(y) \right)$$

$$Slice_H(x) \sqsupseteq (\top, \top), x \in S$$

Definition: $fst((x,y)) = x$ and $snd((x,y)) = y$.

Step 3: Fold $Slice_H$ in the above constraints.

$$Slice_H(x) \sqsupseteq \bigsqcup_{x \to y \in \text{Intra}} Slice_H(y)$$

$$Slice_H(x) \sqsupseteq \bigsqcup_{y \in Summary(x)} Slice_H(y)$$

$$Slice_H(x) \sqsupseteq \lambda z.(fst(z), fst(z)) \left(\bigsqcup_{x \to y \in \text{Call}} Slice_H(y) \right)$$

$$Slice_H(x) \sqsupseteq \lambda z.(\bot, snd(z)) \left(\bigsqcup_{x \to y \in \text{Return}} Slice_H(y) \right)$$

$$Slice_H(x) \sqsupseteq (\top, \top), x \in S$$

Step 4. Although the domain D_H has four values, the function $Slice_H$ can only map to three values $\{(\top,\top),(\bot,\top),(\bot,\bot)\}$, i.e, the function never generates the value $(\top,\bot)$. Therefore, we can redefine the domain

D_H to have the said three values with the following ordering $(\top,\top) \sqsubseteq (\bot,\top) \sqsubseteq (\bot,\bot)$. This redefinition does not cause any loss since it preserves the ordering of elements from the previous definition.

Step 5: The two lambda expressions define functions in the space $D_H \rightarrow D_H$. Give names to the two lambda expression and enumerate the function graph.

$$\begin{aligned}\mu_H &= \lambda x.(fst(x), fst(x)) \\ &= \{(\top,\top) \mapsto (\top,\top), (\bot,\top) \mapsto (\bot,\bot), (\bot,\bot) \mapsto (\bot,\bot)\} \\ \varphi_H &= \lambda x.(\bot, snd(x)) \\ &= \{(\top,\top) \mapsto (\bot,\top), (\bot,\top) \mapsto (\bot,\top), (\bot,\bot) \mapsto (\bot,\bot)\}\end{aligned}$$

Step 6: The domains D_H and D_L are isomorphic. In the above constraints replace each occurance of an element of D_H by the corresponding element of D_L. The resulting constraints are identical, modulo renaming of identifiers, to that of $Slice_L$.